CuEira, Gene-Environment Interaction Analysis on GPU, a first draft

Daniel Berglund

October 2014

Abstract

This is a first draft of a report for a masters thesis in computer science. The aim is to make a program for gene-environment interaction search using GPUs to speed up the calculations. The preliminary results show that there likely is a speed gain when using larger datasets.

Contents

1 Introduction **1**
- 1.1 Outline . . . 1
- 1.2 Genome-wide association studies . . . 1
- 1.3 Genetics . . . 2
- 1.4 Defining Interaction . . . 3
- 1.5 GEIRA, JEIRA and GEISA, the Aim of the Project . . . 3

2 Background **4**
- 2.1 Major, Minor and Risk Allele . . . 4
- 2.2 Algorithmic Background . . . 5
  - 2.2.1 Confounders and Covariates . . . 6
  - 2.2.2 The Multiple Testing Problem . . . 6
  - 2.2.3 Contingency Tables . . . 7
  - 2.2.4 Logistic Regression . . . 8
  - 2.2.5 Data Mining and Machine Learning Approaches . . . 11
- 2.3 Computer Architecture . . . 14
  - 2.3.1 CPU . . . 14
  - 2.3.2 Memory, Caching and Optimizations . . . 15
  - 2.3.3 Concurrency and Threads . . . 18
  - 2.3.4 Accelerators, GPUs and Xeon Phi . . . 20
- 2.4 Performance Measures . . . 24
  - 2.4.1 Amdahl's Law and Gustafson's Law . . . 25
  - 2.4.2 Profilers . . . 27
- 2.5 CUDA programming model . . . 28
  - 2.5.1 Device Memory . . . 30
  - 2.5.2 Streams . . . 31
  - 2.5.3 Efficient CUDA . . . 33
- 2.6 Software Design . . . 36
  - 2.6.1 Unit Tests and Mocks . . . 37
  - 2.6.2 Design Patterns . . . 37
  - 2.6.3 Version Control Software . . . 38

3 Algorithm **39**
- 3.1 Current algorithm, JEIRA and GEISA . . . 39
- 3.2 CuEira . . . 39

4 Results **41**

5 Discussion and Conclusions **42**

6 Outlook **43**

7 Appendix **44**
7.1 List of Variables 44
7.2 List of Abbreviations 44
7.3 Environmental and Covariates File Format 44
7.4 Encountered Bugs 44
7.5 How to Compile CuEira 45

Bibliography **46**

Chapter 1

Introduction

1.1 Outline

The first chapter contains an short introduction to the area and some terminology. The background chapter continues with the terminology and areas related to the project. It starts with some statistics and algorithms and then talks about computer architecture and software design. The algorithm section explains the structure of the current algorithms and then the structure and design of the program that was developed for this thesis, CuEira. After all the background and design are results comparing the performances of the programs and then a discussions of the results.

1.2 Genome-wide association studies

One type of study to find associations between genetic markers and diseases or other traits is genome-wide association studies(GWAS). Most GWAS do not study interactions between the genetic markers or with environmental factors [1, 2]. Investigating gene-gene interactions recently has become more common[1], however gene-environment interactions are still uncommon in studies[2]. Interactions between genes and environmental factors are considered to be important for complex diseases such as cancer and autoimmune diseases[1, 2, 3, 4]. A complex disease develops due to a combination of factors, not a single gene or environmental factor[5]. The environmental factors can be various things such as smoking, physical activity and so on.
GWAS usually has a study design that is either cohort or case-control[5]. In cohort study a sample of a population is followed and over time some individuals will develop the disease of interest[6]. In case-control studies two groups are compared with each other to find risk factors[6]. One group consists of individuals with the disease and the other of individuals that are similar to the cases but who do not have the disease[6].

A typical study consists of tens of thousands of individuals and hundred thousands up to millions of SNPs[1, 7]. Due to the high number of SNPs few algorithms are capable to investigate more than second order interaction in a reasonable time. There are some algorithms that can handle higher interaction orders however these have drawbacks[8, 9, 10].

1.3 Genetics

The genetic information is stored in *chromosomes* each consisting of a *DNA molecule*[11]. Each chromosome comes in a pair where both chromosomes are nearly identical, except for the chromosomes related to sex[11]. Females have two X chromosomes while males have one X and one Y[11]. The DNA molecule is a double stranded helix of four nucleotide bases, *adenine*(A),*cytosine*(C),*guanine*(G) and *thymine*(T). They are always paired as A-T and G-C. The DNA contains many different regions, one type of such regions are *genes*[11]. Genes can *expressed* meaning that they affect the *phenotype*, the pair of genes from both chromosomes is the *genotype*. The phenotype is the result of the genotype, e.g. eye colour, fur patters.

The genetic markers mentioned above are commonly single nucleotide polymorphism(SNP, pronounced snip). SNPs are variations in the genome where a single nucleotide is different between individuals in a population[11].

A position in the DNA is a *locus*[11]. A variation of the same gene or locus is an *allele*[11]. The effect of an gene can be either *dominant*, *recessive* or *co-dominant*. Dominant means that the effect occurs if the gene only needs to be present in one of the chromosomes in the pair, recessive is when the allele has to be present in both chromosomes[11]. Co-dominant is when both genes are expressed, when different alles are present this usually produce some kind of intermediate state[11].

Genotype	Dominant	Recessive
AA	Effect	Effect
GA	Effect	No Effect
GG	No effect	No effect

Table 1.1: *The genetic risk based on the genetic model and if the allele with the effect is A*

1.4 Defining Interaction

There are several ways to define interaction. The overall goal is often to detect if *biological* interactions are present. *Biological* interaction is when the factors co-operate through a physiological or biological mechanism and causes the effect, e.g. the disease. This information can be used to explain the mechanisms involved in causing the disease and possibly help to find cures for them. However biological interaction is not well defined and thus it is not possible to calculate it directly from data.[5, 12]

Statistical interaction on the other hand is well defined. However it is scale dependent, i.e. interactions can appear and disappear based on transformations of the data. Statistical interaction also depends on the model used. The common way to define statistical interaction is to consider the presence of a product term between the variables in the statistical model, this is referred to as *multiplicative* interaction. For instance for a linear model

$$f(x, y) = ax + by + cxy + d \tag{1.1}$$

c is the product term that represents multiplicative interaction between variables x and y. Statistical interaction is often referred to as interaction which can make it a bit confusing.[3, 5]

Additive interaction is a bit broader than multiplicative interaction. It is that the effect from the interaction is larger than the sum of the separate effects. It implies biological interaction as defined by Rothman[5], which is sometimes called *causal interdependence* or *causal interaction*. A more precise definition can be found in section 2.2.4.1

1.5 GEIRA, JEIRA and GEISA, the Aim of the Project

GEIRA is a tool for analysing gene-environment interaction. It uses logistic regression and additive(causal) interaction[3]. *JEIRA* is a parallelised implementation of GEIRA in Java[13]. *GEISA* is based on JEIRA with some changes[14]. These three programs were the basis of this project. The aim of this project was to make the gene-environment interaction analysis faster by using GPUs and to be able to handle larger amounts of data. The program is written in C++ because it is generally faster than Java and CUDA is C/C++. Originally one goal was to implement it using a cluster, but it was cut due to lack of time.

Chapter 2

Background

To search for interaction contains elements of several subjects. This section will go through some of it starting with the algorithms and statistical background. And continuing with computer architecture and software design after that.

2.1 Major, Minor and Risk Allele

The the allele that is least common using all individuals(both cases and controls) is the *minor allele*, the one that is most common is the *major allele*. The frequency of the minor allele is the *minor allele frequency*(MAF)[13, 3]. If the MAF is too low an analysis often will be of little value so it is commonly skipped if it is below a threshold. 5% is common value of the threshold[7, 3].

To determine if the genetic risk is present for each individual one of the alleles needs to be chosen as the *risk allele*. GEIRA and JEIRA determines the risk allele by comparing the MAFs of case and control[14, 13]. If MAF of cases is greater or equal to the MAF of controls the minor allele is used as the risk otherwise the other allele[14, 13].

A simpler equivalent definition is that the risk allele is the allele which frequency is higher in case than in controls. This definition means that the allele chosen as risk might not be the most common in either case or control, however the portion of the population that has the allele is higher among cases than controls. This hints that it might be a reason behind the outcomes. This definition is equivalent to the definition using MAF because if the minor allele has a higher frequency in cases then it the risk is the minor allele according to both definitions. If that is not the case the first definition sets the major allele as the risk. Also due to symmetrical reasons(the total frequency is always 1) this means that the major allele has higher frequency in cases than in controls, which according to the second definition means it is the risk allele.

The presence of genetic risk based on the risk allele and the genetic model can be coded as a variable than then be can used in a statistical model. For dominant or recessive genetic model this means that the variable is binary since the risk is either present or not. See table 1.1 for how the coding is based on the

model. There is an alternative way to code using co-dominant genetic model. In this case the number of risk alleles in the individual used as the variable, so either 0, 1 or 2.

2.2 Algorithmic Background

This section will introduce the statistics and some of the algorithms used to search for interaction. The focus is on logistic regression, however some other methods are briefly examined also.

There are many algorithms and programs proposed for searching for interaction, most have focused on gene-gene interaction as already mentioned[2]. One of the challenges of gene-environment interaction is that environmental factors can be of any variable type(i.e. binary, continuous, categorical) which creates problems in various ways[2]. Gene-gene interaction tools can sometimes be used to find gene-environment interaction, however they usually require the variables to be binary or have other problems since they weren't designed for environmental interaction[2].

Both groups of computers(i.e. clusters) using regular processors[15] and graphic processors[8, 16, 17, 18, 19, 20] have been used for GWAS. Graphic processors for computing have become more popular in the last ten years. They have been a popular choice for some GWAS methods because each combination of variables can commonly be considered independently from the others. More about graphic processors in section 2.3.4 and why the are good for GWAS in section 2.3.4.1.

The methods can be roughly classified into four categories, exhaustive, stochastic, machine learning/data mining and stepwise[10].

Exhaustive search is the most direct approach, it compares all combinations of the SNPs in the dataset. Exhaustive search methods will not miss a significant combination because it didn't consider that specific combination. However it also means that they can be slow since they will spend time on combinations other methods would skip completely. Multifactor-Dimensionality Reduction(MDR)[21] and BOOST[22] are two examples of this type of algorithm.

Stochastic methods uses random sampling to iterate through the data. BEAM[23] is one example and it uses Markov Chain Monte Carlo(MCMC) method.

Data Mining and *Machine Learning* are methods that try to learn patterns from data and tries to generalize it. MDR[21] is a type of data mining method and is among the most common methods used in GWAS. See section 2.2.5 for more details.

Stepwise methods uses a filtering stage and a search stage. At the filtering stage uninteresting combinations are filtered out by using some exhaustive method. The other SNPs are the examined more carefully in the search stage.

BOOST[22] is an example which uses succinct data structures and a likelihood ratio test to filter the data before applying log-linear models.

2.2.1 Confounders and Covariates

Confounding is one of the central issues in design of epidemiological studies. It is when the effect of the exposure is mixed with the effect of another variable. So if we do not measure the second variable, the effect of the first would be estimated as stronger than it really is. The second variable is then a *confounder*. Several methods in epidemiology are about avoiding or adjusting for confounding. Sometimes these variables needs to be incorporated into the models. *Covariates* are possible confounders or other variables that one wants to adjust for in the model. Sometimes covariates are called *control variables*.[12, 5]

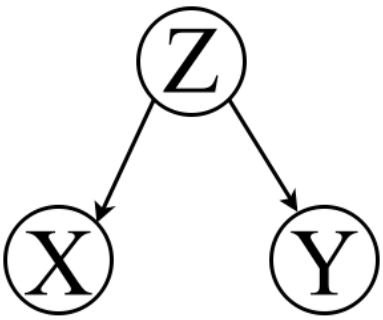


Figure 2.1: *Illustration of a simple case of confounding. If we do not observe Z we might falsely find an association between X and Y*

For instance if a study would look into the effects of yellow teeth on lung cancer they would likely find an effect. However it would be due to the confounder smoking since smoking causes yellow teeth and lung cancer.

2.2.2 The Multiple Testing Problem

It is not uncommon to test many hypothesis on the same data and in the case of GWAS it can be billions of tests. If one continues to test one should eventually find something that is significant due to random chance. With the common significance threshold of 5% it is expected to get 1 in 20 false positives under the assumption that the null hypothesis is true. The problem arises from the fact that the hypothesis tests are dependent on each other since they use parts of the same data. This is the multiple testing problem and if it is not corrected for many false positives might be found.[24]

Bonferroni correction is the simplest method and viewed as a conservative way to correct for this problem. It simply divides the significance threshold by the number of hypothesis tested. However the number of hypothesis made is not always clear. With a two-stage analysis, is the number of hypothesis the number of tests done in both stages combined, the number made in the first stage or the second stage?[24]

2.2.3 Contingency Tables

A contingency table is a matrix used to describe categorical data. Each cell contains a count of occurrences for a specific combination of variables. Table 2.1 is an example of an 2 x 2 table. From this table we can for instance see that 688 smokers got lung cancer. Contingency tables are the basis for various statistical tests to model the data.[25]

	Lung cancer	No lung cancer
Smoker	688	650
Non smoker	21	59

Table 2.1: *Contingency table describing the outcome of a study, from [25], page 42*

2.2.3.1 Relative Risk and Odds Ratio

From a contingency table it is possible to calculate some useful measures such as the risk of getting the outcome based on exposure. The risk for a row i in the table will be referred to as π_i. For table 2.1 the risks π_1 and π_2 is

$$\pi_1 = \frac{688}{688 + 650} = 0.51 \tag{2.1}$$

$$\pi_2 = \frac{21}{21 + 59} = 0.36 \tag{2.2}$$

To compare different risks, for instance between smokers and non smokers, the ratio of the risks is used[25]. It is called *relative* risk(RR) is defined as[25]

$$RR = \frac{\pi_1}{\pi_2} \tag{2.3}$$

So for the table 2.1 the relative risk of getting lung cancer based on exposure to smoking is

$$RR = \frac{0.52}{0.36} = 1.96 \tag{2.4}$$

This means that the risk of getting lung cancer for a smoker is almost twice as high for a non smoker in this data.
Another useful measure is *odds* and *odds ratio*(OR). The odds is[25]

$$\Omega = \frac{\pi}{1 - \pi} \tag{2.5}$$

The odds are non-negative and $\Omega > 1$ when the outcome is more likely than not[25]. So for $\pi = 0.75$ the odds is $\Omega = \frac{0.75}{1-0.75} = 3$. This means that the outcome is 3 times more likely to occur than not. ORs can be used as an approximation of RR in case control studies because RR can not be estimated in that type of studies[26]. Cohort studies on the other hand can give estimates of RR[26]. OR is the ratio of the odds just as RR is the ratio of the risks[25].

$$\theta = \frac{\Omega_1}{\Omega_2} \tag{2.6}$$

In the case when the outcome is a disease or similar variables with odds ratio below one are called *protective* and when it is above one it is a *risk factor*[27].

2.2.4 Logistic Regression

One way to model the contingency tables is by using *logistic regression*. Logistic regression is a type of linear regression model for classification that models a latent probability for the outcomes. The outcomes are binary, however the method can be extended to multiple outcomes. In this work we will only consider them as binary. Logistic regression transforms the probability by using the *logit* transformation. The logit transformation with probability π is [25]

$$\log(\frac{\pi}{1-\pi}) \tag{2.7}$$

The probability with a set of predictor variables X is $\pi(X) = P(Y = 1)$. The linear regression model with n predictors $X = (x_1, x_2, ..., x_n)$, coefficients $\beta = (\beta_1, \beta_2, ..., \beta_n)$ and by using the logit transformation is then[25]

$$logit[\pi(X)] = \alpha + \beta X \tag{2.8}$$

By moving the logit to the right side of the equation we get the model of the probability[25]

$$\pi(X) = \frac{e^{\alpha+\beta X}}{1 + e^{\alpha+\beta X}} \tag{2.9}$$

The logit, equation 2.7, also happens to be the log of the odds(equation 2.5)[25]. By exponentiating both sides of equation 2.8 it shows that the odds is [25]

$$e^{logit[\pi(X)]} = \frac{\pi(X)}{1-\pi(X)} = e^{\alpha+\beta X} = \Omega \tag{2.10}$$

This means that $\exp\beta$ is the odds ratio since the odds increase by $\exp\beta$ for each unit increase of X[25]. It can also been seen by taking the ratio of the odds using equation 2.6 and with $X = x + 1$ and $X = x$

$$\theta = \frac{e^{\alpha+\beta(x+1)}}{e^{\alpha+\beta x}} = e^{\alpha+\beta(x+1)-\alpha-\beta x} = e^{\beta} \tag{2.11}$$

Finding the β coefficients are done in a similar way as with other linear regression models since they all are generalized linear models [25]. It's usually done using maximum likelihood(ML), via Newtons method[25, 13]. It's an iterative method which means it can be relatively slow compared to non iterative methods. The pseudo code for the algorithm using Newtons method can be found in algorithm 1 [13]. * is elemet by element multiplication. X is an NxM matrix that contains the variables, Y is the outcomes with length N.

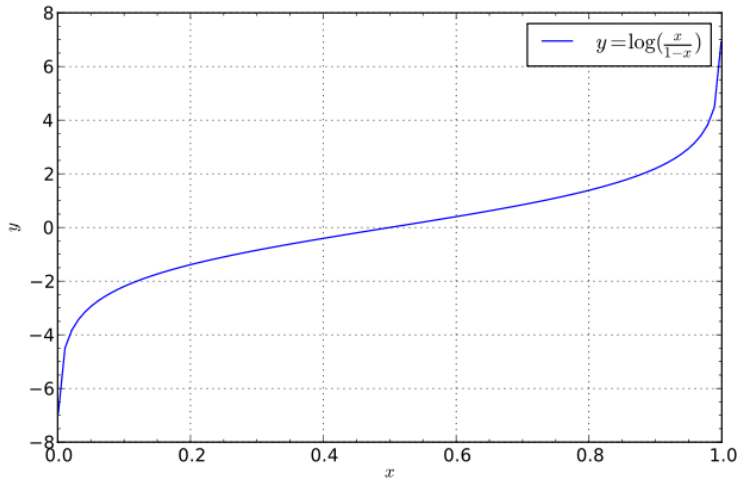


Figure 2.2: *Graph of the logit transformation. Wikipedia Commons*

Algorithm 1 Logistic regression using maximum likelihood and Newtons method

$$\boldsymbol{X} \leftarrow \begin{pmatrix} 1 & \\ \vdots & \boldsymbol{X} \\ 1 & \end{pmatrix}$$
$$\beta \leftarrow \begin{pmatrix} 0 \\ \beta \end{pmatrix}$$
$iter \leftarrow 0$
$diff \leftarrow 1$

while $iter < max\_iter$ ***and*** $diff > threshold$ **do**
  $\beta_{old} \leftarrow \beta$
  $p \leftarrow \frac{e^{\boldsymbol{X}\cdot\beta}}{1+e^{\boldsymbol{X}\cdot\beta}}$
  $s \leftarrow \boldsymbol{X^T} \cdot (\boldsymbol{Y} - p)$
  $\boldsymbol{J} \leftarrow (\boldsymbol{X^T} \cdot (p * (1-p))) \cdot \boldsymbol{X}$
  $\beta \leftarrow \beta_{old} + \boldsymbol{J}^{-1} \cdot s$
  $diff \leftarrow \sum |\beta - \beta_{old}|$
  $iter \leftarrow iter + 1$
end while
$log\ likelihood \leftarrow \sum(\boldsymbol{Y} * lnp + (1 - \boldsymbol{Y}) * ln(1-p))$

2.2.4.1 Mathematical Definition of Interaction for LR

Interaction was talked about in section 1.4. The more precise definition of additive interaction is, the divergence from additive effects on a logarithmic scale, e.g[5].

$$OR_{both\ factors\ present} > OR_{first\ factor\ present} + OR_{second\ factor\ present} - 1 \quad (2.12)$$

To get these specific ORs the variables are coded in the model according to table 2.2[13].

Factor present	First variable	Second variable	Interaction
None	0	0	0
First	1	0	0
Second	0	1	0
Both	0	0	1

Table 2.2: *Coding of the variables for LR*

2.2.4.2 Statistic Measures for interaction

Based on the model and its corresponding ORs there are some measures that can be calculated. These measures show various properties of the additive interaction. They are defined using RR however as mentioned before in section 2.2.3.1 OR can be used to approximate RR.

Relative excess risk due to interaction(RERI) is how much of the risk is due to interaction[27]. It is defined as[27]

$$RERI = RR_{11} - RR_{10} - RR_{01} + 1 \quad (2.13)$$

Attributable proportion due to interaction(AP) is similar to RERI however is the proportion relative to the interaction relative risk[27].

$$AP = \frac{RERI}{RR_{11}} = \frac{1}{RR_{11}} - \frac{RR_{10}}{RR_{11}} - \frac{RR_{01}}{RR_{11}} + 1 \quad (2.14)$$

Synergy index(SI) is the ratio of the combined effects and the individual effects[27].

$$\frac{RR_{11} - 1}{RR_{10} + RR_{01} - 2} \quad (2.15)$$

By using the definition of additive interaction it can be shown that additive interaction is present when $RERI > 0$, $AP > 0$ and $SI > 0$[27].

2.2.4.3 Recoding of Protective Effects

The measures for additive interaction RERI, AP and SI mentioned above are developed for risk factors, i.e. $OR > 1$, which causes problems if a factor is protective[27]. This can be solved by *recoding*, recoding makes sure that no OR is below one[27]. Recoding switches the reference group and the group with the lowest risk[27]. Denoting the different groups with (s, e) where s is the presence of the risk of the SNP variable and e is the presence of the environmental variable. The reference group is $(0, 0)$ and there are three other groups $(1, 0)$,$(0, 1)$ and $(1, 1)$. This means that there are three cases of recoding, one for each group. To recode a group is to transform the variable so that it is changed from protective to risk.

2.2.5 Data Mining and Machine Learning Approaches

Approaches based on Data Mining and Machine Learning have been a popular choice for GWAS. MDR[21] and Random Forest(RF)[28] are among the most common ones[2, 1]. There are other methods as well such as clustering approaches [10]. Most of them are used for screening the data for possible interactions[2, 1].

Their biggest advantage is that they are usually non-parametric and designed with high dimensional data in mind. However they are prone to overfitting and the usual way to try to prevent that is to use cross validation and sometimes permutation tests. It means that even if the method itself is fast it is repeated so many times that the whole algorithm can be slow in the end.[1]

2.2.5.1 Multifactor-Dimensionality Reduction

MDR is a method that reduces the number of dimensions(i.e. variable) by combining several dimensions into one. In GWAS it combines the variables that are suspected to interact. This new variable is then compared against the outcome. If the new variables predictability of the outcome is high enough then the variables that were combined are considered to interact. This process is usually repeated on all pair combinations of variables.

The reduction from n dimensions is done by calculating the ratio of cases versus controls for each combination of the possible values of the variables. If the ratio is above a certain threshold all the members of that groups get the value 1 for the new dimension, otherwise 0. Accuracy of the model is done by using cross validation and permutation tests, in simpler words it means that it reshuffles the data randomly and recalculates the model many times to get an estimate of the models certainty. Because of that MDR can be slow. However it is still usually faster than exhaustive search with regression methods.[1, 21]

MDR can been used for gene-environment interaction but requires modifications since MDR can only handle binary variables. There are extensions that can use continues variables, however these are regression based so these will be slower than regular MDR.[2]

A simple example of MDR using exclusive or (XOR). XOR is an logical operator that is true if one and only one of its two variables is true. We have 4 possible combinations and an occurrence for each of them, see table 2.3. The combination (1,0) and (0,1) both have one case with outcome 1 so MDR will classify them as 1 in the new variable Z, the other two combinations have outcome 0 so will be classified with Z=0, see table 2.4. From here it is easy to make an predictor from Z to the outcome Y by comparing the values.

Y	$\mathbf{X_1}$	$\mathbf{X_2}$
1	1	0
1	0	1
0	0	0
0	1	1

Table 2.3: *XOR table with outcome* **Y** *and variables* $\mathbf{X_1}$ *and* $\mathbf{X_2}$.

Y	**Z**
1	1
1	1
0	0
0	0

Table 2.4: *XOR table with* $\mathbf{X_1}$ *and* $\mathbf{X_2}$ *combined into* **Z** *using MDR.*

2.2.5.2 Random Forest

RF is an ensemble learning method[28]. Ensemble methods combine multiple models to improve performance. RF takes randomized samples of the data and builds decision trees on each of them. These trees are then combined to form the classifier. Usually hundreds or thousands of trees are used depending on the problem[28]. One of the most popular variants of Random Forest for GWAS is Random Jungle[29].

It has been shown in high dimensional data that RF tends to only rank interacting factors high if they have strong marginal effects[30]. Also the ranking of the variables does not indicate which factor it is interacting with either since it is based on the joint distributions[2]. How to incorporate the environmental factors in RF is also not obvious and using variables with very different scales can bias RFs results[2].

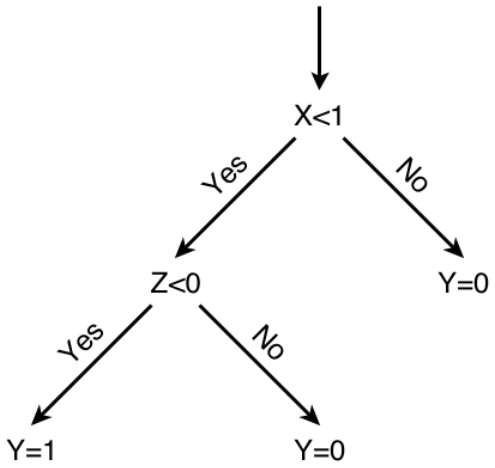


Figure 2.3: *A example of a decision tree*

2.3 Computer Architecture

This section will explain some of the architectures used to perform tasks in a computer. A large part is focused on optimization techniques that are in used today. For the purpose of this thesis it is easiest to think about the computer as three main parts, data storage(e.g. the disk), processor and the memory. Sometimes there is also an accelerator, for instance a graphic processor, which will be explained in section 2.3.4.

2.3.1 CPU

Central processing unit(CPU) is the part of the computer that performs most of the tasks[31]. It executes instructions one by one in its core. Most modern CPUs have multi-core architecture. These CPUs have multiple cores[31]. Each core can perform tasks independent of each other so the programs needs to be parallel to get maximum speed. Parallel means that the program issues multiple instructions at the same time, more about this in section 2.3.3.

The figures 2.4 and 2.5 shows how the CPU is divided into areas and that most of it is not used for calculations. A large part of the area is used for various optimizations. The next section 2.3.2 will explain some of these optimizations.

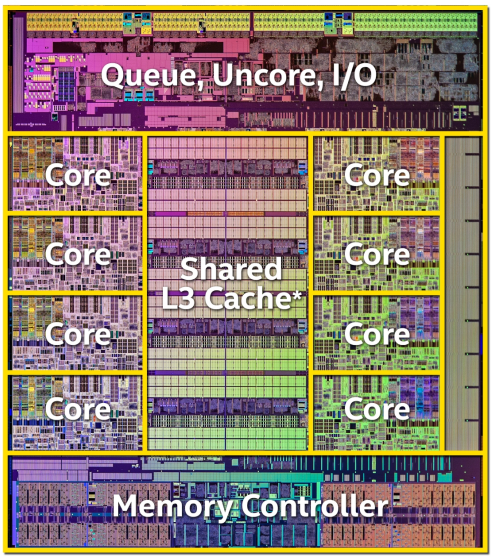


Figure 2.4: *The layout of the Haswell architecture i7-5960x, from [32]*

2.3.2 Memory, Caching and Optimizations

Various optimizations have been introduce to the CPUs over the years[33, 31]. Some are common and almost all CPUs have them, others are unique to a specific vendor or CPU[31]. In this section some of the more common optimizations will be explained. As shown in figure 2.5 these optimizations take up the most of the cores area, the square in the upper left corner executes the instructions.

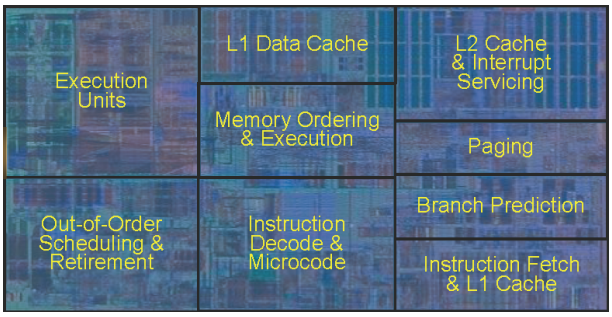


Figure 2.5: *Layout of a CPU core, Intel Nehalem i7, from [34]*

Retrieving data from memory is relatively slow compared to how fast the processor works[31, 33]. To solve this problem a *cache* was introduced[33]. It stores recently used data in a very small and very fast memory that resides on the CPU chip itself[33]. The caches varies in sizes, the latest ones are around 16-20mb, for instance Intels i7-5960x with 20mb [35]. The cached data can then be reused fast without waiting for the main memory to fetch it. However when the data is not in the cache it takes the usual speed to fetch it from the main memory, that is a *cache miss*[33]. Avoiding cache misses is important for speed because of how much time it can take to fetch the data from the main memory[33]. Modern CPUs commonly have more than one cache, most have three[31]. They are named L1 to L3, with L1 being the one fastest and smallest and L3 the largest and slowest[31].

Modern multicore processors several L1 caches, one for each core[31]. It is commonly one L2 per core too, however it is shared in some architectures[31]. L3 is almost always shared between all cores[31]. That multiple caches are used creates a problem called *cache coherency*. It is that when the data is modified in one cache all the cores caches needs to be updated too so that they use the new value[31].

Another memory optimization somewhat related to caches is *prefetching*[31, 33]. Instead of fetching just the data requested by the current operations it also fetches the surrounding data[33]. The chance that the surrounding data will be used soon is high since it is common to iterate over arrays and similar structures[31]. If a program iterates over an array unless it was used recently the first value will be a cache miss since it is not in the cache. However the following values will be due to them being prefetched and cached. At some point however the array is too long to be prefetched and cached completely and when the CPU hits that spot a new cache miss will happen[33]. Because everything is stored sequentially it is important in some cases to know how the it is stored so that

the prefetching and caching can be used. One of these cases is for matrices, they are 2D but stored as one array. Different programming languages store them differently, either column by column or row by row. Row by row is called *row major* and column by column is *column major*.

In section 6.2.1 in What Every Programmer Should Know About Memory[33] an example of how much speed that can be gained by optimizing matrix matrix multiplication. The first version is shown in algorithm 2.

Algorithm 2: *Matrix matrix multiplication*

```
for (i = 0; i < N; ++i)
  for (j = 0; j< N; ++j)
    for (k = 0;k < N; ++k)
      res[i][j]+= mul1[i][k] * mul2[k][j];
```

By transposing the mul2 matrix it becomes more cache friendly because it uses the prefectching of surrounding data the way that mul1 already is[33]. The transpose itself can be eliminated while also making it use the cache better by reading in the correct amount of data, this is the size SM in the code[33]. The optimized version is shown in 3. It took 17.3% of the original time[33].

Algorithm 3: *Matrix matrix multiplication optimized*

```
#define SM (CLS / sizeof (double))

for (i = 0; i < N; i += SM)
  for (j = 0; j < N; j += SM)
    for (k = 0; k < N; k += SM)
      for (i2 = 0, rres = &res[i][j], rmul1 = &mul1[i][k]; i2 <
          SM; ++i2, rres += N, rmul1 += N)
   for (k2 = 0, rmul2 = &mul2[k][j]; k2 < SM; ++k2, rmul2 += N)
     for (j2 = 0; j2 < SM; ++j2)
       rres[j2] += rmul1[k2] * rmul2[j2];
```

On top of that by using some additional features of the the CPU do multiple instructions at once further speed is gained increasing it to 9.47% of the original time[33].

If the operation instructions diverge, e.g. if statements, the CPU would have to wait for the results to see what instructions load and do next[33]. This can mean a significant loss of time, however thanks to a optimization called *branch prediction* and *speculative execution* that can be prevented in some cases[33]. A dedicated part of the CPU stores the results from these diverges and when they are repeated it uses them to make an guess of what do to next[33]. It then loads the instructions and tries to fetch the needed data[33]. It then speculatively executes the instructions, i.e it executes instructions that might not actually be needed[33]. If the guess was correct the results are kept, on the other hand if it was wrong the CPU starts again from the correct path and discards the incorrect results[33].

Sometimes there can be several independent instructions, e.g. when adding two vectors together, this allows optimizations for *instruction level parallelism*[31]. The order of the instructions can be ignored which for instance means that if an instruction needs to wait for some data another instruction that has its data available could be executed while waiting[31]. It is an optimization called *out of order execution*[31]. There are also other possible optimizations[31].
In the example 1 the instructions are independent because they use different variables so they can be executed in any order. If the variables b and c needs to be fetched but e and f are available due to previous instructions then the CPU could execute the second addition first by using out of order execution. However if an operation later depends on the variable a or d then that instruction would have to wait until the additions have been executed.

Example 1: *Independent Instructions*

```
a = b + c;
d = e + f;
```

Pointer aliasing can prevent the compiler from making certain optimizations, e.g. out of order execution mentioned above. Pointer aliasing is when the same memory area can be accessed using different variables[31]. This becomes a problem if an set of instructions use these variables in the same area of the program. The relative order of operations between these operations then has to be preserved. The compiler does have the information if aliasing occurs or not which means the compiler has to treat all situations where it can occur as if it does[31].

If the integers in example 1 are changed to pointers then aliasing can occur because some of these pointers could point to the same memory. For instance if a and e points to the same memory then the result would be wrong if the second line is executed before the first. This would force the instructions to be executed in order or the result would likely be incorrect.

Example 2: *Pointer Aliasing*

```
//All variables are pointers
*a = *b + *c;
*d = *e + *f;
```

Some programming languages, e.g. Fortran, disallow some types of aliasing while others, e.g. C/C++, allows aliasing[31]. The compilers for the languages in the second case commonly have an option to disable aliasing. However for languages that don't allow aliasing, or if it is disabled, it is then the programmers responsibility to make sure that aliasing does not occur or the results can be wrong[31].

2.3.3 Concurrency and Threads

Concurrency means doing multiple things and the same time, instead of *sequential*. This can cause various problems such as the same object being accessed at the same time. The dinning philosophers is an concurrency problem that can used to illustrate some of these problems. A group of philosophers is sitting at a round table, each has a plate of spaghetti in front of them and there is a fork between each pair of philosophers[36]. The philosophers alternate between thinking and eating. However they can only eat if they have both the right and left fork[36].

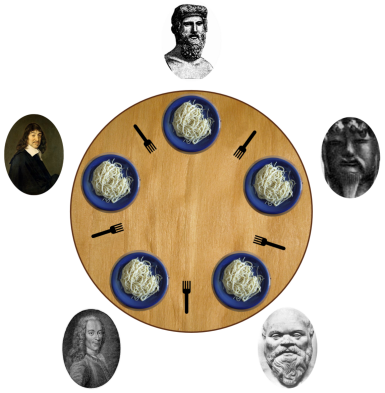


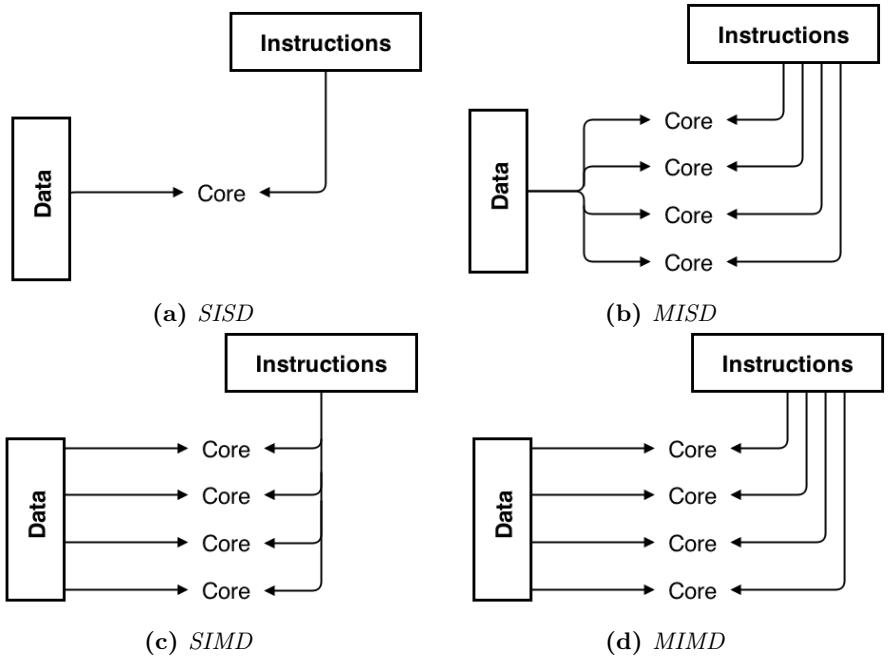
Figure 2.6: *Illustration of the dining philosophers problem. Wikipedia Commons*

A potential proposal[36] for behaviour instructions could be:

- Think
- Wait for a fork to become available and pick up that fork
- Wait for and pick up the other fork
- Eat
- Put down the forks one by one
- Go back to thinking

The problem is that this set of instructions can lead to a state where everyone is holding one fork and waiting to get the other one[36]. But no one is done eating so no forks will become available. The system is then locked into its state, this is called a *deadlock*[36, 31]. A potential solution to the problem in this case is to introduce a person that dictates if a philosopher is allowed to eat or not[36].

There are also other potential problems that can arise when using concurrency. A *race condition* occurs when the result depends on what affects it first, it is a race between them on who can reach it first[31]. *Locks* can be used to prevent situations as the ones described above by making sure only one thread at a time can access the object[31].



(a) *SISD* **(b)** *MISD*

(c) *SIMD* **(d)** *MIMD*

Figure 2.7: *Flynns taxonomy of parallel architectures[31]*

In computer science concurrency occurs when instructions are separated in different *threads*. The architectures can be divided in a taxonomy with four categories including the sequential. They are illustrated in figure 2.7.

- *SISD*(Single-Instruction,Single-Data) is the sequential and is as the name says, single instructions on single data[31].
- *SIMD*(Single-Instruction, Multiple-Data) is applying the same instruction on different data[31]. SIMD is also called vectorisation[31].
- *MIMD*(Multiple-Instruction, Multiple-Data) applies different instructions on different data[31].
- *MISD*(Multiple-Instruction, Single-Data) performs different operations at the same piece of data, this paradigm is uncommon[31, 37].

2.3.4 Accelerators, GPUs and Xeon Phi

Accelerators are separate parts made to do one task and do it well. They are usually highly specialised and perform badly outside of the tasks they were designed for. The two accelerators explained here is graphical processing unit(GPU) and Intel Xeon Phi[38, 39]. GPUs have mostly been used for games and other graphics related problems however have become more popular for general computing the last 10 years due to their high number of cores[38]. Intel Mic is much more recent with prototypes in 2010 and the first generation released in 2010[39]. It is Intels response to the GPUs and has elements of both GPU and CPU[39]. This section will focus on explaining some parts of the architecture of the GPUs and Xeon Phi. How to use the GPUs are talked about in more detail in section 2.5.

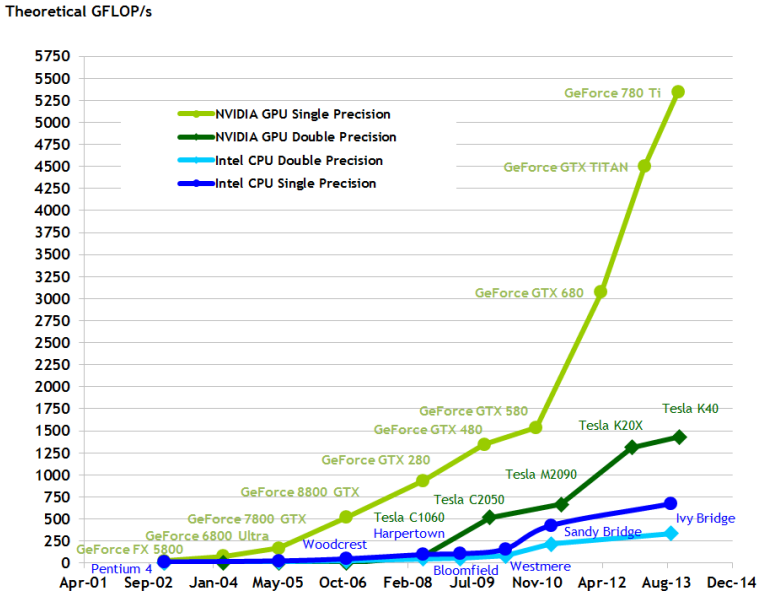


Figure 2.8: *Theoretical throughput of some NVIDIA GPUs and Intel Processors, from[38]. However the chart does not contain a Xeon Phi processor.*

GPUs are made to process images in a fast manner so that games and similar tasks runs smoothly[38]. Because GPUs are made for image processing they are GPUs have a large number of cores(1000-2000)[38, 41]. Each core is slower than a core in a CPU, however the number of cores makes up for it. The downside is that it's slower than a normal CPU for sequential tasks so the program has to take advantage of the extreme parallelism to get good speed[42]. The memory of the GPU is separate from the computers main memory which means that the data needs to be transferred between the memories which increases the programs complexity[38].

The GPU cores work in a manner that is similar to SIMD[38]. The cores also have fewer optimizations than CPUs, this means that they can then devote a larger portion of the chip to the calculations as seen in figure 2.9. For instance NVIDIAs GPUs lack optimizations such as branch prediction and spec-

ulative execution talked about in section 2.3.2 [38]. Their caches are also much smaller than the CPUs[38]. Another inheritance from the image processing is that they are much faster with single precision than with double precision datatypes[38, 41]. A CPU drops slightly in performance when using double precision but much less than a GPU as seen in figure 2.8. Single and double precision refers to how many bytes are used to store numbers, fewer bytes means less numerical precision.
Intels answer to the increasing popularity of GPUs for general computing is Xeon Phi[39]. It has elements of GPU architecture however is still more like a CPU. It has separate memory and many cores(50+)[39]. It is MIMD which can be good for codes that diverge a lot, for ease of use however it has the possibly to use more SIMD like structures[39]. It is possible to only use the Xeon Phis memory and avoid the memory transfers that way. The memory is relatively small compared to the main memory so if the program needs a lot of memory transfers are likely needed. The cores of the Xeon Phi lacks some of the normal optimizations, e.g. out of order execution[39].

2.3.4.1 GPU vs Xeon Phi, Why use GPUs for Interaction?

What is most efficient depends largely on the algorithm and if there is any previous existing code. For instance if the algorithm contains many points of divergence Xeon Phi is likely faster because Xeon Phi is MIMD while GPU is SIMT. Xeon Phi can be applied to a broader set of problems due to using a MIMD architecture[39]. A one on one comparison of speed might not be entirely relevant because the algorithm used for the test might simply be better suited for one the architectures. Speed per cost, either power consumption or the price on the hardware, can also be more relevant. If one is half the cost of the other then comparing two versus one would be a better comparison. CUDA is also more mature than Xeon Phi since Xeon Phi is only a few years old while CUDA is almost ten[38, 39].

So what suits best needs to be looked at using a case by case basis by examining the specifics of the algorithm and what type of computers that might already be available. It is also important to think about how important maximum speed is and how much time there is to spend. If extremely high speed is of importance and time is not a concern then a GPU is likely to get higher performance, unless the algorithm is badly suited for GPUs. However Xeon Phi might reach similar speed but with less effort.

GPUs are good for interaction algorithms because most methods consider each combination independently and also in this case that LR consists of linear algebra operations. A new combination can be transferred while another is calculated. Linear algebra with sufficiently large matrices and vectors work very well on GPUs[44, 38]. Several studies got high gains from implementing their algorithms on GPU[8, 16, 17, 18, 19, 20]. However most studies had CPU versions that were not parallel so it is likely that the gains would have been less if they would have been compared to an optimized and parallel CPU version, especially if a Xeon Phi co-processor was used. One study did make a comparison of a CPU cluster version and a GPU version of their algorithm, they found that 16 CPU nodes had the same performance as a single GTX 280 card[45].

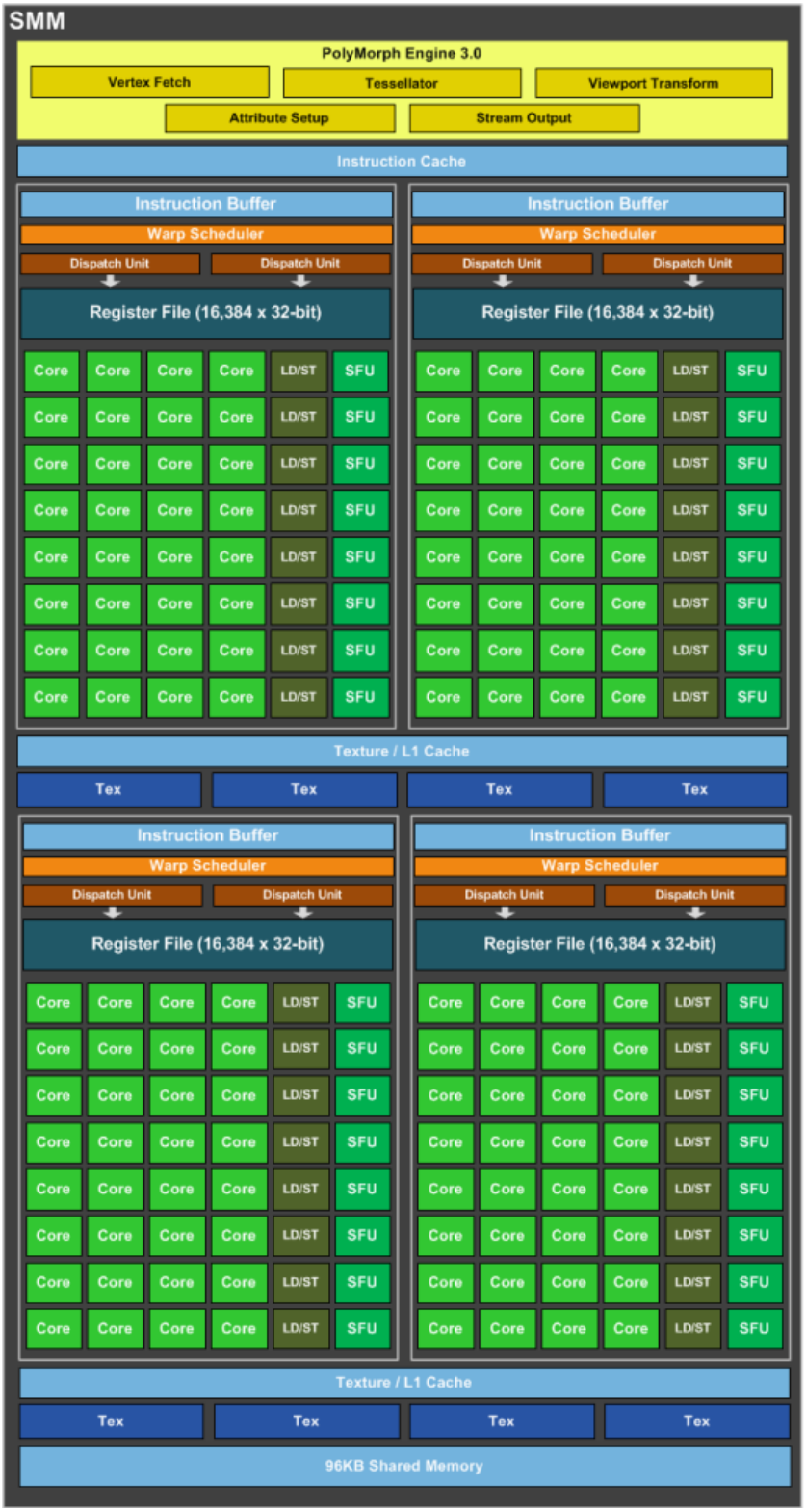


Figure 2.9: *Layout of the Maxwell architecture, from [40]*

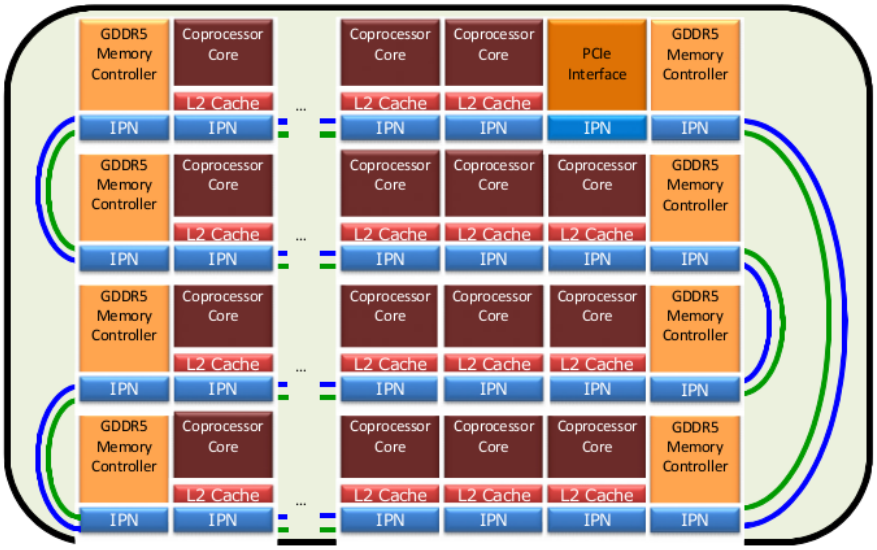


Figure 2.10: *Knights Corner silicon layout, from[43]*

2.4 Performance Measures

An important part in making fast and efficient programs is to know how fast the program is under certain conditions and which parts of the program are slow[31]. For instance the speed could suddenly drop when we start using too many threads, there might be a bottleneck in the network, and so on.

There are two ways to measure how long a program takes to execute[31]. Wall clock time is how long real life time the program took. The other is to measure the number of processor cycles spent. A parallel program will have shorter execution time than it is serial version however it will likely have spent more processor cycles due to overhead from communication and initialization of the threads. These two measures are useful for different kinds of comparisons. Wall clock time is better for overall performance while number of cycles is useful for comparing different algorithms[31, 42].

Speed up is a measure of how much faster then program is with a certain number of threads compared to the serial version. It's defined as[31]

$$S(p) = \frac{T(1)}{T(p)}$$

Where $T(1)$ is execution time of serial program and $T(p)$ is execution time of parallel program with p threads. Linear speed up is when S(p)=p[31].

Efficiency reflects how efficient the program is using p threads. Linear speed has efficiency 1. It's defined as[31]

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)}$$

Strong scaling refers to how the program handles a fixed problem size and increased number of processors[31]. An program with strong scaling has linear speed up[31]. Weak scaling refers to the execution time of the program when there is a fixed problem size *per processor* and the number of processors is increased[31, 42].

It can be a good idea to plot these measures while varying p, this can show when a bottleneck occurs. Looking at the measures at node level can also be useful to get an idea of how increased number of nodes and therefore increased communication over the network affects the performance.

2.4.1 Amdahl's Law and Gustafson's Law

Amdahl's Law is used to find the expected speed up of a system when parts of it are made concurrent. Simply it says that as the number of processors increases the parts that aren't parallel will start taking up more and more of the wall clock time and that the speed up for adding more processors will decrease as more and more processors are added and more time is spent relatively on the non parallel parts[31]. It's closely related to strong scaling.[42, 46]

It says that the expected speed up with F fraction of the code parallel and p threads is[31]

$$S(p) = \frac{1}{(1-F) + \frac{F}{p}(1-F)}$$

As the number of threads grow towards infinity $S(p)$ converges on $\frac{1}{1-F}$. If we have 90% of a code parallel then even with infinite number of threads we won't get a better speed up than ten.[46]

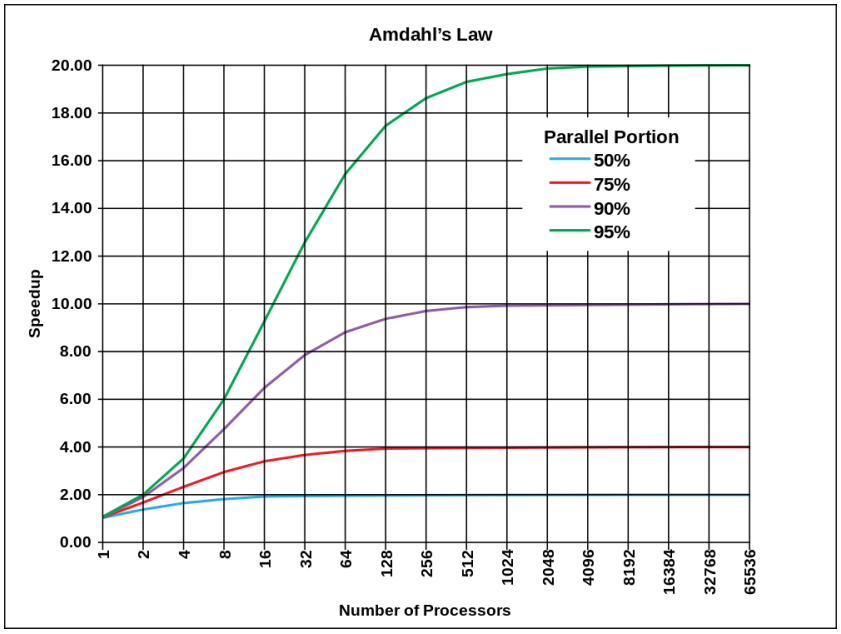


Figure 2.11: *Illustration of Amdahl's Law. Wikipedia Commons*

There are limitation to Amdahl's Law since it makes a couple of assumptions.

- The number of executing threads remain constant over the course of the program.
- The parallel portion has perfect speed up. Often not true due to shared resources,eg caches, memory bandwidth, and shared data.
- The parallel portion has infinite scaling, not true due to similar limits as above. More threads will not increase performance after a while or might even decrease it.

- There is no overhead for creation and destruction of threads.
- The length of the serial portion is independent of the number of threads. Often the serial work is to divided the work to the threads, this work will obviously increase as the number of threads go up. More threads can also lead to more communication overhead.
- The serial portion can't be overlapped by the parallel parts. For instance with producer consumer type pattern the consumer could be strictly serial but the time it takes could be overlapped by the parallel producers.

This means it is most accurate with programs that are of the fork-join type, e.g. both serial and parallel parts[47].

Gustafson's Law is closely related to Amdahl's Law and can in some ways be more accurate than Amdahl's Law[47]. Gustafson's Law makes similar assumptions as Amdahl's Law however it also makes two additional statements. It states that problems tends to expand when provided with more computational power, e.g. increased precision by reducing grid size for simulations, higher frame rate for graphics and so on[47, 31]. The second is that the parallel portion of the program tends to expand faster than the serial part, e.g. for matrix multiplication the initialization scales linearly with the matrix size while the multiplication itself scales as $O(n^3)$[47, 31]. The former means that it is closely related to weak scaling[31]. So in a way it says that the execution time remains constant rather than the amount of data. More precise it says that the expected speed up with p threads and F fraction of the code that is parallel is[47, 42, 31]

$$S(p) = p + (1 - F)(1 - p)$$

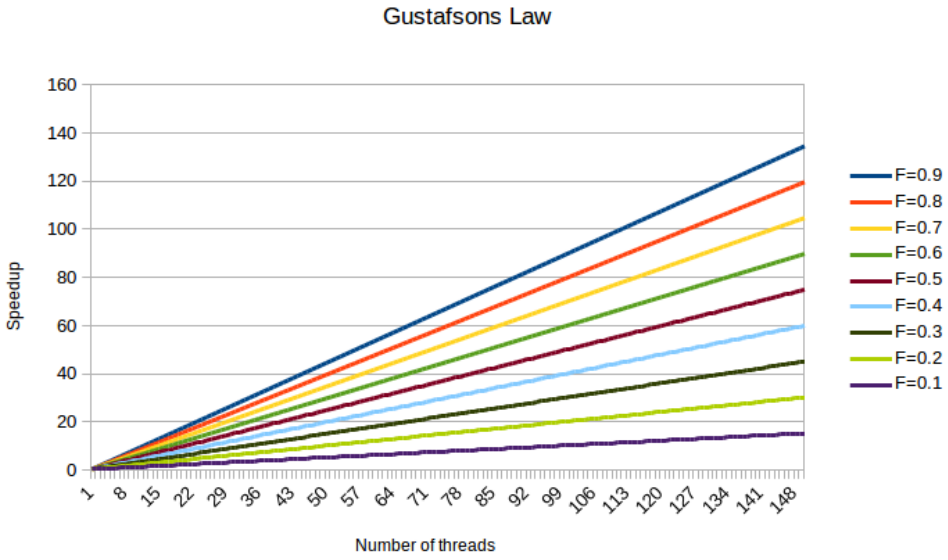


Figure 2.12: *Illustration of Gustafson's Law*

2.4.2 Profilers

There are applications called profilers that are made to assess the programs performance and resource consumption. They calculate some of the measures mentioned earlier and they also check hardware usage and how much time the program spends at various parts of the program. This is very useful for finding bottlenecks and other problems in the program. It does not matter if the algorithm is super fast if all the data is stuck in network transfers. The profilers can be hardware dependent so the manufactures usually provided them for their products.[31, 42]

2.5 CUDA programming model

This section is about GPU programming in more detail using NVIDIAs CUDA(Compute Unified Device Architecture)[38]. CUDA extends C and C++ with some additional functionality that can be used to perform operations on NVIDAs GPUs. It provides a separate compiler to compile the GPU code called nvcc[38]. Different GPUs support different CUDA functions, each NVIDIA GPU has a value called computational capability. The higher the value the more features of CUDA are supported on that GPU. Some properties also vary between the different underlying architectures. This means that it is important to know what kind of capability the GPU to be used has so that the program does not need features that are not there. The things described here might not apply to GPUs with compute capability below 3. There are GPUs made specifically for calculations rather than games. This section is an summary of how CUDA works from the CUDA programming guide[38] and CUDA best practices guide[42].

In CUDA the GPU is called *device* and systems CPU and memory is the *host*. To perform operations on the device a type of function called *kernel* is used. A kernel is works mostly as a normal C/C++ function with the addition of some specifiers to provided options to set number of threads and so on. Example 3 is an example of a kernel and how it can be called from the host code. The kernel add each element of the arrays A and B storing it in C. Each thread performs on addition and knows why elements to use based on it's id. The call to the kernel is made by giving three arrays and the number of threads to be used as N. In this case N needs to be the length of the arrays.

The ___global___ keyword in front of the function declaration tells nvcc that it is a kernel. The $<<< N, M >>>$ specifier tells the compiler how many N blocks and how many M threads per block that the kernel should use[38]. A block is a group of threads and shares some memory and resources. The blocks can be executed in any order so they need to be completely independent from each other but variables can be shared among threads inside a block[38]. This allows the program to scale to different GPUs as shown in figure 2.13. The blocks are organized in a one, two or three dimensional *grid*. These can be accessed by each thread so it knows which grid it is in as illustrated in figure 2.14. This can be used to make it easier to assign the threads to the correct bit of the calculation. For instance using a two dimensional grid is good for matrices[38, 42].

Example 3: *Simple Kernel*

```
// Kernel definition
__global__ void Add(float* A, float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

// In the host code
// Kernel invocation with N threads
Add<<<1, N>>>(A, B, C);
```

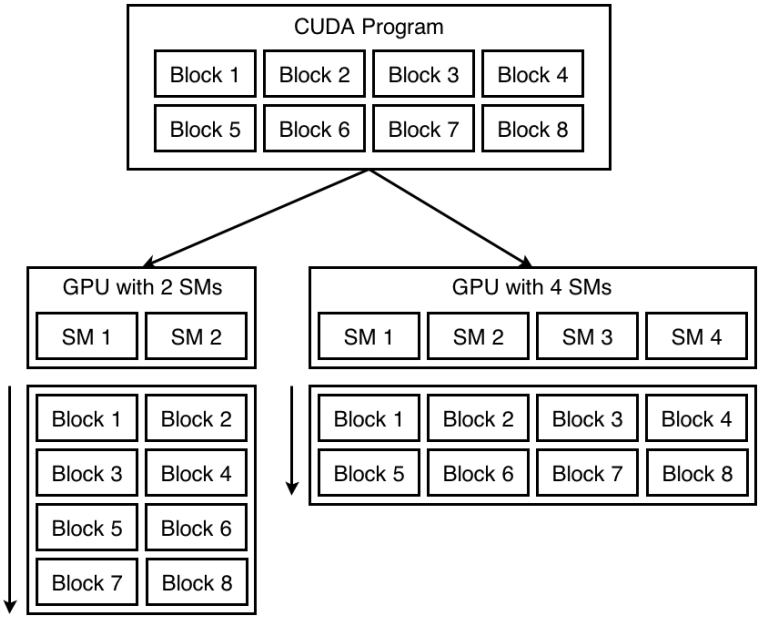


Figure 2.13: *Blocks execution depending on the number of streaming multiprocessors*

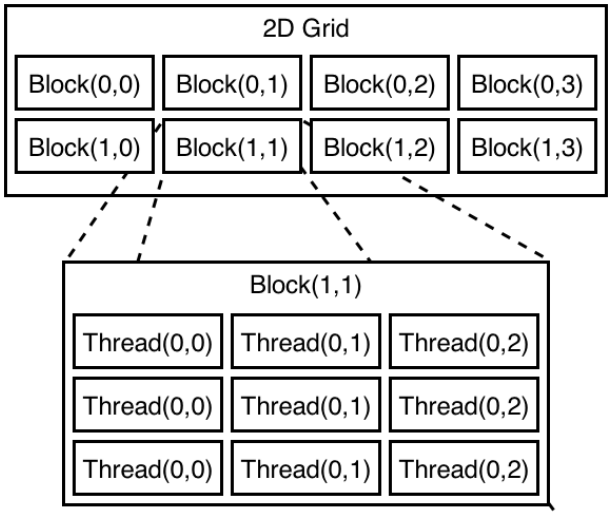


Figure 2.14: *2D grid of blocks*

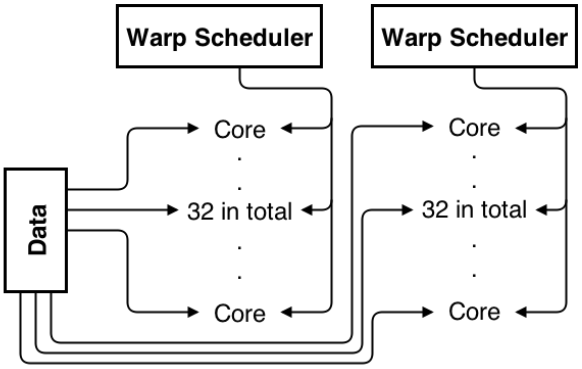


Figure 2.15: *SIMT architecture*

The streaming multiprocessors(SM) is the hardware that handles the execution of these blocks and can execute hundreds of threads concurrently. It uses *SIMT*(Single-Instruction, Multiple-Thread) which is similar to SIMD described earlier in section 2.3.3. SIMT works mostly as SIMD except that it can act as MIMD on collections of threads called *warps*. Each warp consists of 32 threads. When an SM gets a block it is split into warps that are assigned to warp schedulers. Each warp scheduler gives one instruction to a warp so full efficiency is achieved when all the 32 threads perform the same instruction. If there is any divergence it has to disable unrelated threads, so divergence can be costly. However different groups of warps are on different warps schedulers so can diverge without problem.[38]

2.5.1 Device Memory

The GPUs memory is physically on a different device separate from the computers main memory which means that they have separate memory spaces. An object in one memory is not accessible in the other memory. The computers main memory is the *host memory* and the GPUs memory is the *device memory*. Since they are separate data has to be transferred to the device memory and this is done by explicit calls to transfer sections of the hosts memory.[38]

The GPU also have several different types of memory[38]. Correct usage can give increased speed[38, 42].

- Register memory is located on the multiprocessor and usually costs zero cycles to access. The multiprocessor splits the available registers over its threads so if there are many threads that uses many variables not all of them will fit in the register. This is why a program sometimes can be faster with lower number of threads.[38]
- Global memory is the main memory of the GPU and is accessible from all threads and blocks. However it is relatively slow to access.[38]
- Shared memory is shared inside a block and is faster than global memory. However it is limited in size.[38]
- Constant memory is small however it is read only which enables some optimizations. It is best used for small variables that all threads access.[38]

- Local memory is tied to the threads scope, however it still resides off-chip so it has the same access time as global memory.[38]
- Texture memory is read only and can be faster to access than global memory in some situations. This was more important in older GPUs when global memory was not cached.[20, 38]
- Read-only cache is available on GPUs based on Kepler architecture and uses the same cache as the texture memory. The data as to be read only each multiprocessor can have up to 48kb of space depending on GPU.[48]

2.5.2 Streams

The kernel calls can be made on a *stream*. The easiest way to think of the stream is as queue. The kernels on a stream will be executed in the order they are made but kernels from different streams can be executed in any order given there is enough computational resources. In this way it is possible to execute up to 32 concurrent kernels depending on GPU.[38]

This is not entirely true for some GPU architectures because they only have one queue for the kernels. Because of this independent kernels from different streams can block each other. If we have a group of kernels and issue them on stream one first and then on stream two all the kernels in stream two will be stuck in the queue waiting for the kernels in stream one to finish. This is because the kernels queue only looks at the first kernels to see if they can be executed. The second kernel in the first stream then blocks the queue from moving forward so it will not see the kernels on stream two that could be executed. Newer architectures after Kepler on the other hand has several queues so they don't have this problem.[38, 42, 48]

Streams are also important for asynchronous transfers. These transfers are executed on a stream and just as kernels gets executed after the previous kernels on the same stream is done. The advantage is that other streams can do calculations as normal while the transfer happens. This can hide the time for transfers completely in some situations as shown in 2.17. However the host memory has to be pinned, pinned memory means that the operative system can not page that memory. Paging is that the operative system stores a part of the memory in another area, usually the disk memory, to save space in the memory. Too much pinned memory can slow down the computer.[38, 49]

However there is problem that asynchronous transfers can cause depending on the GPUs architecture[49, 50]. Some older GPUs only have one copy engine while newer have one for each direction, one to the GPU and one from the GPU[49, 50]. This can affect how the calls should be ordered and using the wrong one can make the performance worse than without using asynchronous transfers[49, 50].

There is an example of this in [50] which illustrates the problem. They have four versions of the same code, a sequential transfer version and three asynchronous transfer versions. Two different GPUs were used, one had one copy engine the other had two. In the asynchronous the data is split over four streams coloured differently in the figure 2.17. Version 1 initiates the calls by looping over the streams one by one and doing the transfer and kernel calls on that stream before moving on to the next[50]. Version 2 makes all the host to device transfer calls for all streams first, then the kernels and then the device to host transfer call[50]. Version 3 is the same as version 2 but with a dummy even after each kernel[50]. The figure 2.17 shows how the transfers and kernels are executed on the GPU.

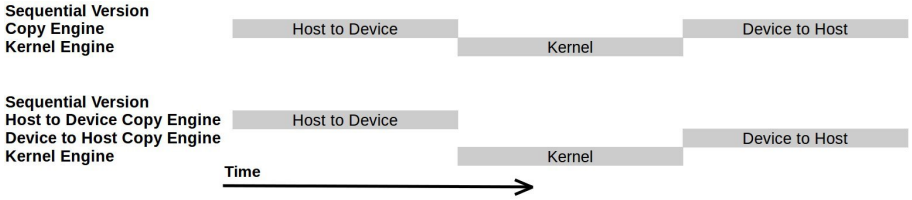


Figure 2.16: *Sequential versions*

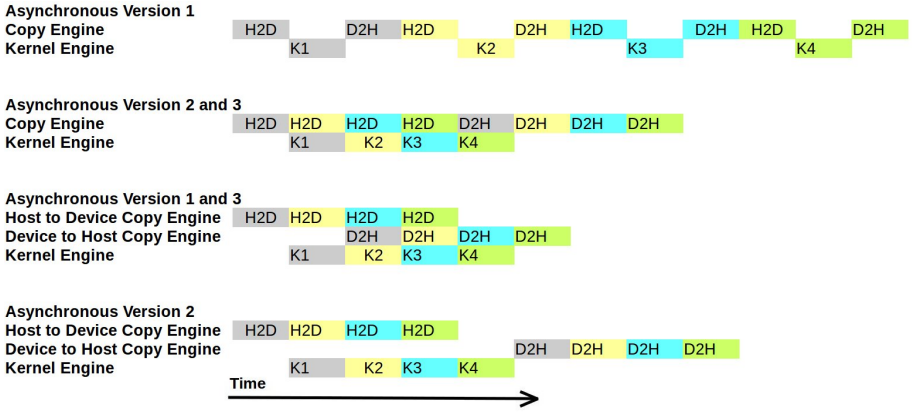


Figure 2.17: *Asynchronous versions. D2H=device to host transfer. H2D=host to device transfer. Each colour represents a stream.*

2.5.3 Efficient CUDA

One of the main criticism against GPUs for general computing purposes is that it is hard to get good performance because it requires good knowledge about details of the GPU architecture, especially the memory architecture. This section is a summary of things that can be good to consider for CUDA programs from the CUDA programming guide[38], CUDA best practices guide [42] and a similar list as this one, however slightly outdated now since it is from 2010, from a master thesis about GPU in GWAS[20].

Maximize parallelism
Structure the program and the algorithm in such a way that it is as parallel as possible and overlap the serial parts on CPU with calculations on the GPU.[20, 38]

Minimize transfers between host and device
Moving data between host and device is expensive and should be avoided if possible. It can be better to run serial parts on the GPU rather than moving the data to the host to do the calculation on the CPU. The bandwidth between host and device is one of the large performance bottlenecks. This can be a problem when the data is to large to fit in the relatively small GPU dram.[38, 42]

Find the optimal number of blocks and threads
There are many things affected by the number of blocks and threads so they should be considered carefully. It is a good idea to parametrize them so that they can be changed for future hardware and varied for optimization. NVIDIA has an occupancy calculator which can be helpful in determining the optimal numbers, however high occupancy does not mean high performance.[38, 42]

The number of blocks should be larger than the number of multiprocessors so that all multiprocessors have at least one block to execute. Having two blocks or more per multiprocessor can be good so that there are blocks that aren't waiting for a ___syncthreads() that can be executed. However this is not always possible due to shared memory usage and similar.[42]

The number of threads per block should be a multiplier of 32 but minimum 64. It's also important to remember that multiple concurrent blocks can reside on the same multiprocessor. Too large number of threads in a block and parts of the multiprocessor might be idle since there aren't a block small enough to use those threads. Between 128 and 256 threads is a good place to start.[42]

Use streams and asynchronous transfers
By using streams it is possible to overlap memory transfers with calculations as mentioned before. This means that the data for the next batch can be transferred while the current batch is calculated and when it is done it can start calculating on the next batch directly after the current one is done. This can hide the time for transfers completely in some sit-

uations. Depending on the time the transfers take versus the time the calculations take this can give significant speedup.[38, 49, 48]

Use the correct memory type and caches
Correct use of caches and memory is important for both CPU[33] and GPU. However it is more complicated on GPU since the caches are smaller and there are several types of memory as mentioned before[38, 42].

Avoid divergence
Each thread in a warp executes the same instruction at the same time so if some of threads diverge the rest will be ideal until they are at the same instruction again. This means it is important to use control structures such as if statements carefully to prevent threads from idling.[38, 42]

Avoid memory bank conflicts when using shared memory
Shared memory is divided into equally-sized memory modules called banks that can be accessed at the same time for higher bandwidth. Bank conflicts occur when separate threads access the same bank. On some GPUs it is fine if all threads access the same bank. Bank conflicts are split into as many conflict-free requests as needed.[38, 42]

Use existing libraries
Instead of writing everything from scratch it is usually a good idea to use already existing libraries. Especially when performance is important and most task are non trivial on GPUs so using an already optimized library is a good idea. Some of the most popular libraries for CUDA are:

- CUBLAS: BLAS implementation for CUDA. BLAS and LAPACK is a standard for a library that provides highly optimized functions for linear algebra.[44]
- CULAtools: BLAS and LAPACK implementation for CUDA for both dense and sparse matrices[51]
- MAGMA: BLAS and LAPACK implementation among other things that can distribute the work on both CPU and GPU[52]
- Thrust: Template based library that tries to emulate C++ standard library[53]

Avoid slow instructions
There are some instructions that can be slow and should be avoided if possible, for instance type conversion, integer division and modulo. If a function is called with a floating point number that might be used as a double and require a conversion. By putting an f at the end of the number it is told to be single precision float, for instance 0.5f. In some cases it is possible to use bitwise operations instead which is faster.[38, 42]

Restricted pointers can give increased performance
Aliasing can be a problem as mentioned earlier. By using the ___restrict___ keyword on pointers the compiler can be told that no aliasing will occur, however it is up to the programmer to make sure that is the case or there might be unexpected results. Not using aliasing reduces the number of memory accesses the CPU needs to make. However it increases register pressure so it can have an negative effect on performance.[38]

Use fast math functions if precision isn't needed

There are three versions of the math functions. The double precision version is func() while the single precision function is funcf(). There is third faster but less accurate version _funcf(). The option -use_fast_math makes the compiler change all the funcf() to _funcf().[42]

Instruction level parallelism can increase speed

Just as for CPUs the GPUs can take advantage of instruction level parallelism. By unrolling loops this can give 2x the speed relatively simple[54].

2.6 Software Design

How to design software so it can be maintained over time has been a problem for a long time. Object oriented languages such as C and later Java and C++ arose because of problems with maintaining software. Badly organized and written code will cost productivity in the future when bugs and other problems stack up because of earlier mistakes. Fixing those bugs are likely to time consuming because it can be hard to find where they originated from. All code gather problems overtime, however a well designed system will degenerate significantly slower than one that was designed carelessly. The code will also be read by ourselves and others later which means readability is important if a future reader is to understand the code and find the parts they are interested in. [55, 56]

This is also important in science where others might wish to use the tools or repeat the experiments. A recent study[57] found that the repeatability in computer science is low. They used the term reproducibility however repeatability is a more fitting term because they didn't try to reproduce the results from scratch, they tried to repeat it using the same code. Of 513 articles they received the code from 231, 108 failed to build, 21 failed to run properly[57]. Just 102 worked properly[57].

There are ways to design the program and code in such a way that it is easier to read and maintain. Most of the things described here goes under the development technique called agile development and the concept of clean code and is a summary of some concepts from the book Clean Code[55]. SOLID is an acronym for five principles for object oriented programming and design[55]. When used together they are intended to make programs that are easy to maintain and extend[55]. As already mentioned readability is important. Correct names will make the code explain itself without other documentation, the code itself is the documentation[55].

Initial	Principal	Concept
S	Single responsibility principle	A class should only have a single responsibility
O	Open/closed principle	A class should be open for extension but closed for modification
L	Liskov substitution principle	If S is a subtype of T, then objects of type T may be replaced by objects of type S without without altering any of the desirable properties of the program(e.g. work performed)
I	Interface segregation principle	Use several smaller and more specific interfaces instead of one large
D	Dependency inversion principle	Depend on abstractions(e.g. interfaces) not details

Table 2.5: *The five SOLID principles, from [55]*

Dependency injection is one way to implement dependency inversion principle[55]. Injection is passing the dependency to the dependent object. This is used instead of allowing the dependent object to construct or find the dependency.

2.6.1 Unit Tests and Mocks

Unit testing involves testing the program in small units in isolation. Testing in isolation means that the test should only depend on the part of the program that is tested. If a part of the program is not working properly only its related tests should fail, not other tests for code that depends on it but otherwise works properly. This makes it easier to find errors when they do occur since the tests will pinpoint the unit which does not work.[55]

It can be easy to denote test code as less important than the main code but they should be treated as equally important. Tests should also not be an inconvenience to use so they should be easy to run, take reasonable amount of time to complete and not require any outside interpretation whether they failed or not.[55]

Mocking is to replace a real object with a fake object called a *mock* that for the code is indistinguishable from the real object. This allows one to create situations and test with more control and without depending on the real objects code. The second is important for unit tests since it enables one to test units that normally depend on others in isolation. In the first case it is useful in situations such as when one wants to test a class handling of a rare failure. Such failures can be hard and time consuming to induce. It is then easier to use a mock object that behaves like the failure has occurred. However mocking requires that the code for the class doesn't create the object itself since there is no way to replace the object with the mock. This is one of the reasons why dependency injection should be use.[55]

2.6.2 Design Patterns

A *design pattern* in software design is a reusable general solution to a common recurring problem in a given context. It is templates and structures to solve the problem, however it is not code. However they are partially dependent on the programming language since different languages have different features and limitations.[56]

Consumer producer is a concurrency pattern for when there are a number of consumers/workers and producers. They share a common queue for products. The producer generates some product and put it into the queue, while the the consumers consume the products. The problem is that the producers should not add to an non empty slot and that each product should only be consumed once.[55, 56]

2.6.3 Version Control Software

Backups for data, code, text and so on is important if something happens. One way to backup text and code projects is to use a version control software and is usually an important tool in developing[58]. For this project Git was used and the source code can be found at `https://github.com/Berjiz/CuEira`. Version control software allows several developers to share a common *repository* which allows them to work on different parts at the same time, it could even be in the same file[58]. However changes at the same place causes *merge conflicts* which usually needs to be solved manually[58]. A version control software keeps track of all the changes made so if a part later turns out to be wrong that part can reverted to an older correct version[58]. Version control software also have many other features[58].

Chapter 3

Algorithm

3.1 Current algorithm, JEIRA and GEISA

JEIRA and GEISA uses Java and its built-in functions for concurrency. Since GEISA is based on JEIRA, as talked about in section 1.5, the underlying design is the same.

The implementation uses the producer consumer pattern described in section 2.6.2. The main thread creates a queue of tasks. All the result producers iterates over this shared queue and outputs results. These results are placed in a queue a consumed by another thread that prints the results. The program reads all data at the start which means it can get memory problems for large datasets.

3.2 CuEira

CuEira(CUDA Environment Risk Analyser) was designed and developed as a part of this thesis. It is written in C++. Information about dependencies and how to compile can be found in appendix 7.5.

The program has two steps, an initialisation step and a calculation step. In the initialisation step it reads all the data except the information for the individuals in the SNPs, i.e. it doesn't read the bed file at that stage. For the calculations the main thread starts a number of threads, one for each CUDA stream. The number of streams depends on how many GPUs the computer have, CuEira starts three streams per GPU. The threads share the data that has been read in the initialisation, among them is an queue of the SNPs. The queue has an lock to prevent more than one thread at a time to fetch a SNP.

Each thread asks the queue for a SNP, reads its information and launches the calculations. When its done it prints the results and asks the queue for a new SNP until the queue is empty. The calculations are done mostly on GPU however some small parts are done on the CPU. This was done for two reasons, to not to do small taks on the GPU and prevent it from doing larger kernels instead. When thread reaches parts where it mostly use the CPU another thread

can use the GPU, this one of the reasons to use streams. Secondly one of the operations(the singular value decomposition, SVD) was not available in CUBLAS and turned out to be tricky to do. The matrix in question is small so it was decided to do the SVD operation using the CPU where a library was available. A future examination where more these parts are done on GPUs to would be interesting and could increase performance by minimizing transfers.

Chapter 4

Results

Some first preliminary results on a small dataset with 2000 individuals and 100 SNPs and a larger dataset with 3446 individuals and 133 648 SNPs. GEISA is much faster than CuEira on the small dataset. CuEira takes 12.17 seconds to complete using 3 streams on 4 C2050 GPUs with the calculations taking 0.3 seconds. GEISA takes 2.6 seconds with calculations taking less than a second. A closer look at CuEira reveals that a part in the reading of the input files takes 10 seconds.

The larger dataset took 6 minutes and 28 seconds for GEISA, with 5 minutes and 24 seconds spent on calculations. CuEira takes 3 minutes and 55 seconds, with calculations taking 3 minutes and 40 seconds. The slow part mentioned in the earlier paragraph took 10.8 seconds. However the numbers for CuEira on this dataset should be taken with a grain of salt. CuEira skips SNPs with missing data currently while GEISA does not. A dataset without missing data will be used for the final report, together with some variations in size.

The data structure for the individuals in CuEira likely needs to be changed, it would likely cost much less time as a vector instead of a hashmap. However it seems to remain fairly constant on 10 seconds so might not be a problem for large datasets.

Chapter 5

Discussion and Conclusions

Chapter 6

Outlook

Chapter 7

Appendix

7.1 List of Variables

θ_{ij} Odds ratio with exposure levels i and j

RR_{ij} Relative risk with exposure levels i and j

X Predictors

Y The outcome, binary variable

$\pi(x) = P(Y = 1)$

Ω Odds

7.2 List of Abbreviations

MAF
: Minor allele frequency, the frequency of the least common allele.

RF
: Random Forest

7.3 Environmental and Covariates File Format

The environmental factors and covariates are stored in separate files with the same format, one file for environmental factors and one for the covariates if any. One of the columns needs to contain the individual ids, the rest of the columns should be data. The delimiter can be anything of the usual ones and there is an option to provided it to the program. The default is tab delimited.

7.4 Encountered Bugs

This is a small collection of various bugs and limitations encountered in the course of the project that can be good to know for anyone that wants to compile CuEira or if the libraries are used in other projects.

There are several problems with Intel compilers and libraries. Most libraries work fine with gnu C/C++ compiler but have problems with Intel compilers. Boost 1.55 does not work with Intel compilers, 1.54 works. AllOf in Google Test also does not work together with Intel compilers.

Nvcc can not compile c+11 code, no interface included in the code that nvcc compiles can use c++11. This problem can largely be avoided by using separate compilation. The CUDA find package module for CMake(included in version 2.8 and later) has a default flag that should be removed as it can cause problems. The flag tells nvcc to propagate the host code flags, if the compiler for the host code then uses c++11 that flag is propagated to nvcc and it will not compile The flag is disabled in CMake by using set(CUDA_PROPAGATE_HOST_FLAGS off).

7.5 How to Compile CuEira

The source code for the program is available at github, `https://github.com/Berjiz/CuEira`.

List of dependencies. Listed version is the version used.

- Boost, tested with version 1.54, 1.55 does not work due to a bug with Intel compilers
- CMake, version 2.8
- CUDA, version 5.5
- MKL, version 13.1
- Intel compiler version 14.0
- Google Test and Google Mock, already included in the folder.

First install the dependencies, they might need to be compiled from source. Open a terminal and make a new directory where you want the program to be compiled. Enter the directory and write:
cmake /path/to/CuEira/
Then type:
make

The program should now be compiling. CMake might complain about not finding a dependency. If it does you need to tell CMake where it is. The path to boost is set by using EXPORT BOOST_ROOT = /path/to/boost/. The path to the compilers is done with CXX = cmake /path/to/source for the C++ compiler and C = cmake /path/to/source for the C compiler.

Bibliography

[1] H. J. Cordell, "Detecting gene–gene interactions that underlie human diseases," *Nature Reviews Genetics*, vol. 10, no. 6, pp. 392–404, 2009.

[2] S. J. Winham and J. M. Biernacka, "Gene–environment interactions in genome-wide association studies: current approaches and new directions," *Journal of Child Psychology and Psychiatry*, vol. 54, no. 10, pp. 1120–1134, 2013.

[3] B. Ding, H. Källberg, L. Klareskog, L. Padyukov, and L. Alfredsson, "Geira: gene-environment and gene-gene interaction research application," *European Journal of Epidemiology*, vol. 26, no. 7, pp. 557–561, 2011.

[4] H. Mahdi, B. A. Fisher, H. Källberg, D. Plant, V. Malmström, J. Rönnelid, P. Charles, B. Ding, L. Alfredsson, L. Padyukov, *et al.*, "Specific interaction between genotype, smoking and autoimmunity to citrullinated α-enolase in the etiology of rheumatoid arthritis," *Nature genetics*, vol. 41, no. 12, pp. 1319–1324, 2009.

[5] K. Rothman and S. Greenland, *Modern Epidemiology*. London: Lippincott Williams and Wilkins, 1998.

[6] C. Mann, "Observational research methods. research design ii: cohort, cross sectional, and case-control studies," *Emergency Medicine Journal*, vol. 20, no. 1, pp. 54–60, 2003.

[7] P. R. Burton, D. G. Clayton, L. R. Cardon, N. Craddock, P. Deloukas, A. Duncanson, D. P. Kwiatkowski, M. I. McCarthy, W. H. Ouwehand, N. J. Samani, *et al.*, "Genome-wide association study of 14,000 cases of seven common diseases and 3,000 shared controls," *Nature*, vol. 447, no. 7145, pp. 661–678, 2007.

[8] B. Goudey, D. Rawlinson, Q. Wang, F. Shi, H. Ferra, R. M. Campbell, L. Stern, M. T. Inouye, C. S. Ong, and A. Kowalczyk, "Gwis-model-free, fast and exhaustive search for epistatic interactions in case-control gwas," *BMC genomics*, vol. 14, no. Suppl 3, p. S10, 2013.

[9] G. Fang, M. Haznadar, W. Wang, H. Yu, M. Steinbach, T. R. Church, W. S. Oetting, B. Van Ness, and V. Kumar, "High-order snp combinations associated with complex diseases: efficient discovery, statistical power and functional interactions," *PloS one*, vol. 7, no. 4, p. e33531, 2012.

[10] S. Leem, H.-h. Jeong, J. Lee, K. Wee, and K.-A. Sohn, "Fast detection of high-order epistatic interactions in genome-wide association studies using information theoretic measure," *Computational Biology and Chemistry*, 2014.

[11] D. Sadava, D. Hillis, C. Heller, G. Orians, and W. Purves, *Life The Science of Biology*. Sinauer Associates, 8th ed., 2008.

[12] K. J. Rothman, *Epidemiology: an introduction*. Oxford University Press, 2002.

[13] D. Uvehag, "Design and implementation of a computational platform and a parallelized interaction analysis for large scale genomics data in multiple sclerosis," 2013.

[14] D. Uvhage and H. Zazzi, "Geisa." `https://github.com/menzzana/geisa`, 2014.

[15] A. Gyenesei, J. Moody, A. Laiho, C. A. Semple, C. S. Haley, and W.-H. Wei, "Biforce toolbox: powerful high-throughput computational analysis of gene–gene interactions in genome-wide association studies," *Nucleic acids research*, vol. 40, no. W1, pp. W628–W632, 2012.

[16] L. S. Yung, C. Yang, X. Wan, and W. Yu, "Gboost: a gpu-based tool for detecting gene–gene interactions in genome–wide case control studies," *Bioinformatics*, vol. 27, no. 9, pp. 1309–1310, 2011.

[17] Z. Zhu, X. Tong, Z. Zhu, M. Liang, W. Cui, K. Su, M. D. Li, and J. Zhu, "Development of gmdr-gpu for gene-gene interaction analysis and its application to wtccc gwas data for type 2 diabetes," *PloS one*, vol. 8, no. 4, p. e61943, 2013.

[18] S. Lee, M.-S. Kwon, I.-S. Huh, and T. Park, "Cuda-lr: Cuda-accelerated logistic regression analysis tool for gene-gene interaction for genome-wide association study," in *Bioinformatics and Biomedicine Workshops (BIBMW), 2011 IEEE International Conference on*, pp. 691–695, IEEE, 2011.

[19] S. Chikkagoudar, K. Wang, and M. Li, "Genie: a software package for gene-gene interaction analysis in genetic association studies using multiple gpu or cpu cores," *BMC research notes*, vol. 4, no. 1, p. 158, 2011.

[20] J. Poznanovic, "Plinkgpu: A framework for gpu acceleration of whole genome data analysis," Master's thesis, School of Informatics, University of Edinburgh, 2010.

[21] M. D. Ritchie, L. W. Hahn, N. Roodi, L. R. Bailey, W. D. Dupont, F. F. Parl, and J. H. Moore, "Multifactor-dimensionality reduction reveals high-order interactions among estrogen-metabolism genes in sporadic breast cancer," *The American Journal of Human Genetics*, vol. 69, no. 1, pp. 138–147, 2001.

[22] X. Wan, C. Yang, Q. Yang, H. Xue, X. Fan, N. L. Tang, and W. Yu, "Boost: A fast approach to detecting gene-gene interactions in genome-wide case-control studies," *The American Journal of Human Genetics*, vol. 87, no. 3, pp. 325–340, 2010.

[23] Y. Zhang and J. S. Liu, "Bayesian inference of epistatic interactions in case-control studies," *Nature genetics*, vol. 39, no. 9, pp. 1167–1173, 2007.

[24] J. M. Bland and D. G. Altman, "Multiple significance tests: the bonferroni method," *Bmj*, vol. 310, no. 6973, p. 170, 1995.

[25] A. Agresti, *Categorical data analysis*. John Wiley & Sons, Second ed., 2002.

[26] H. T. O. Davies, I. K. Crombie, and M. Tavakoli, "When can odds ratios mislead?," *Bmj*, vol. 316, no. 7136, pp. 989–991, 1998.

[27] M. J. Knol, T. J. VanderWeele, R. H. Groenwold, O. H. Klungel, M. M. Rovers, and D. E. Grobbee, "Estimating measures of interaction on an additive scale for preventive exposures," *European journal of epidemiology*, vol. 26, no. 6, pp. 433–438, 2011.

[28] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[29] D. F. Schwarz, I. R. König, and A. Ziegler, "On safari to random jungle: a fast implementation of random forests for high-dimensional data," *Bioinformatics*, vol. 26, no. 14, pp. 1752–1758, 2010.

[30] S. J. Winham, C. L. Colby, R. R. Freimuth, X. Wang, M. de Andrade, M. Huebner, and J. M. Biernacka, "Snp interaction detection with random forests in high-dimensional genetic data," *BMC bioinformatics*, vol. 13, no. 1, p. 164, 2012.

[31] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[32] Intel, "Intel unleashes its first 8-core desktop processor." `http://newsroom.intel.com/community/intel_newsroom/blog/2014/08/29/intel-unleashes-its-first-8-core-desktop-processor`, 2014.

[33] U. Drepper, "What every programmer should know about memory," 2007.

[34] F. Abi-Chahla, "Intel core i7 (nehalem): Architecture by amd?." `http://www.tomshardware.com/reviews/Intel-i7-nehalem-cpu,2041-2.html`, 2008.

[35] Intel, "i7-5960x specification sheet." `http://ark.intel.com/products/82930`, 2014.

[36] C. Hoare, *Communicating sequential processes*. Prentice-Hall, Inc., 1985.

[37] W. F. Gilreath and P. A. Laplante, *Computer Architecture: A Minimalist Perspective: Dynamics and Sustainability*. Springer, 2003.

[38] NVIDIA, *NVIDIA CUDA C Programming Guide*, v5.5 ed., July 2013.

[39] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[40] M. H. NVIDIA, "Maxwell: The Most Advanced CUDA GPU Ever Made." http://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/, 2014.

[41] NVIDIA, "Nvidia tesla-kepler product description." http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf.

[42] NVIDIA, *NVIDIA CUDA C Best Practices Guide*, v5.5 ed., July 2013.

[43] Intel, "Intel Xeon Phi Coprocessor DataSheet, Document ID 328209 003EN." http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html, 2014.

[44] NVIDIA, *CUBLAS Library User Guide*, v5.5 ed., July 2013.

[45] R. Jiang, F. Zeng, W. Zhang, X. Wu, and Z. Yu, "Accelerating genome-wide association studies using cuda compatible graphics processing units," pp. 70–76, 2009.

[46] X.-H. Sun and Y. Chen, "Reevaluating amdahlâĂŹs law in the multicore era," *Journal of Parallel and Distributed Computing*, vol. 70, no. 2, pp. 183–188, 2010.

[47] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[48] NVIDIA, "Kepler Tuning Guide." http://docs.nvidia.com/cuda/kepler-tuning-guide.

[49] Mark Harris, "How to Overlap Data Transfers in CUDA C/C++." http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/.

[50] G. Ruetsch and B. Leback, "CUDA Fortran Asynchronous Data Transfers." http://www.pgroup.com/lit/articles/insider/v3n1a4.htm.

[51] "CULA Tools: GPU Accelerated Linear Algebra," 2010.

[52] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–8, IEEE, 2010.

[53] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010. Version 1.7.0.

[54] V. Volkov, "Unrolling parallel loops," *Tutorial at the*, p. 133, 2011.

[55] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008.

[56] R. C. Martin, "Design principles and design patterns." http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf, 2000.

[57] G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren, “Measuring reproducibility in computer systems research.” `http://reproducibility.cs.arizona.edu/v1/tr.pdf`, 2013.

[58] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development.* "O'Reilly Media, Inc.", 2012.